Visual Debugging of Dataflow Systems

FANTI MACHMOUNT AL SAMISTI
Visual Debugging of Dataflow Systems

Fanti Machmount Al Samisti

Master of Science Thesis

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

14 August 2017

Examiner: Associate professor Jim Dowling
Supervisor: Theofilos Kakantousis

TRITA-ICT-EX-2017:152
Abstract

Big data processing has seen vast integration into the idea of data analysis in live streaming and batch environments. A plethora of tools have been developed to break down a problem into manageable tasks and to allocate both software and hardware resources in a distributed and fault tolerant manner. Apache Spark is one of the most well known platforms for large-scale cluster computation. In SICS Swedish ICT, Spark runs on top of an in-house developed solution. HopsWorks provides a graphical user interface to the Hops platform that aims to simplify the process of configuring a Hadoop environment and improving upon it. The user interface includes, among other capabilities, an array of tools for executing distributed applications such as Spark, TensorFlow, Flink with a variety of input and output sources, e.g. Kafka, HDFS files etc.

Currently the available tools to monitor and instrument a stack that includes the aforementioned technologies come from both the corporate and open source world. The former is usually part of a bigger family of products running on proprietary code. In contrast, the latter offers a wider variety of choices with the most prominent ones lacking either the flexibility in exchange for a more generic approach or the ease of gaining meaningful insight except of the most experienced users.

The contribution of this project is a visualization tool in the form of a web user interface, part of the Hops platform, for understanding, debugging and ultimately optimizing the resource allocation and performance of dataflow applications. These processes are based both on the abstraction provided by the dataflow programming paradigm and on systems concepts such as properties of data, how much variability in the data, computation, distribution, and other system wide resources.
Sammanfattning


De verktyg som finns för att övervaka den tidigarenämnda teknologi-stacken kommer från både företag och öppna källkod projekt. Den tidigare är vanligtvis en del av en större familj med produkter som kör på proprietär kod. I kontrast mot den senare, som erbjuder en större mängd med val där de viktigaste har bristande flexibilitet i utbyte mot ett mer generiskt tillvägagångssätt eller enkelhet att få nyttig information förutom för de mest erfarna användarna.

Bidraget från det här projektet är ett visualiseringsspråk i form av ett webbanvändargränssnitt, integrerat med Hops plattformen, för förståelse, felsökning och i slutändan kunna optimera resursallokering och prestanda för dataflödesapplikationer. Dessa processer är baserade på både abstraktionen från dataflöde programmerings paradigmen och på systemkoncept såsom dataegenskaper, datavaribiabilitet, beräkning, distribution och andra systemegenskaper.
This work would not be possible without the invaluable help of my supervisors Gautier Berthoug and Theofilos Kantousis with their knowledge and patience for all my questions. Jim Dowling, other than being my examiner, provided much needed guidance to this project. Last but not least, thank you to the lab colleagues and friends that made the every day lab sessions a funnier place to be.
# Contents

1 Introduction
   1.1 Problem description ................................... 2
   1.2 Problem statement ..................................... 3
   1.3 Goals .................................................. 4
   1.4 Reflections of Ethics, Sustainability and Economics .... 4
   1.5 Structure of this thesis .............................. 5

2 Background
   2.1 Hadoop .................................................. 8
   2.2 HDFS ........................................................ 9
      2.2.1 NameNode ......................................... 9
      2.2.2 DataNode ......................................... 11
   2.3 YARN ................................................... 11
      2.3.1 Resource Manager ................................. 12
      2.3.2 Node Manager ................................... 14
      2.3.3 Application Master .............................. 14
   2.4 Data computation ................................... 15
      2.4.1 MapReduce ..................................... 15
      2.4.2 Limitations and criticism ...................... 18
      2.4.3 Spark ........................................... 18
   2.5 Telegraf .............................................. 22
   2.6 Graphite ............................................. 23
   2.7 InfluxDB ............................................. 24
   2.8 Related work ........................................ 25

3 Methods .................................................. 27
   3.1 Data collection & analysis ................................ 28

4 Implementation ......................................... 31
   4.1 Collecting metrics ..................................... 31
   4.2 Database schema ..................................... 35
List of Figures

1.1 Big data tools of Apache Software Organization . . . . . . . . . . 2
2.1 HPC architecture cluster overview . . . . . . . . . . . . . . . . 8
2.2 Hadoop high level architecture . . . . . . . . . . . . . . . . . 9
2.3 HDFS architecture . . . . . . . . . . . . . . . . . . . . . . . . 10
2.4 YARN RM and NM interactions . . . . . . . . . . . . . . . . . 12
2.5 RM components . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.6 MapReduce architecture . . . . . . . . . . . . . . . . . . . . . . 16
2.7 Spark platform components . . . . . . . . . . . . . . . . . . . . 20
2.8 A spark lineage graph . . . . . . . . . . . . . . . . . . . . . . . 21
2.9 Spark’s distribution of tasks . . . . . . . . . . . . . . . . . . . . 22
2.10 Graphite high level architecture . . . . . . . . . . . . . . . . . 24
4.1 Metrics gathering overview . . . . . . . . . . . . . . . . . . . . . 32
4.2 Architectural overview . . . . . . . . . . . . . . . . . . . . . . . 41
4.3 Frontend architecture . . . . . . . . . . . . . . . . . . . . . . . 43
4.4 Sequence diagram of a controller . . . . . . . . . . . . . . . . . 44
4.5 Overview of a streaming application’s state . . . . . . . . . . . . 48
5.1 Events investigation and tuning . . . . . . . . . . . . . . . . . . 52
List of Tables

2.1 Comparison of Spark monitoring tools . . . . . . . . . . . . . . . 26
4.1 Metrics from /allexecutors per executor . . . . . . . . . . . . . . . 42
5.1 Spark Pi 30.000 iterations, Dynamic execution on . . . . . . . . . 53
5.2 Spark Pi 30.000 iterations, Fixed # executors, 2 cores . . . . . . . 53
Listings

4.1 Spark metrics settings ........................................ 33
4.2 Excerpt from Hadoop nodemanager settings ............. 34
4.3 Simple AngularJS controller ................................. 40
# List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator [1]</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System [2]</td>
</tr>
<tr>
<td>RM</td>
<td>ResourceManager</td>
</tr>
<tr>
<td>NM</td>
<td>NodeManager</td>
</tr>
<tr>
<td>AM</td>
<td>ApplicationMaster</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>NN</td>
<td>NameNode</td>
</tr>
<tr>
<td>DN</td>
<td>DataNode</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RDD</td>
<td>Resilient Distributed Dataset</td>
</tr>
<tr>
<td>HOPS</td>
<td>Hadoop Open Platform-as-a-Service</td>
</tr>
<tr>
<td>HOPSWORKS</td>
<td>User interface for the HOPS platform</td>
</tr>
<tr>
<td>DTO</td>
<td>Data Transfer Object</td>
</tr>
<tr>
<td>SVG</td>
<td>Scalable Vector Graphics</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The notion of information processing and analysis is not a new term, seeing it’s first appearance in the mid of 19th century. Scientists projected on the sizes of the libraries in the upcoming years or the growth of academic papers and journals. This trend has only been increasingly spreading throughout our daily lives due to the deep embrace of big data processing and analysis through advertisements, targeted marketing etc. Among others, Facebook serves on a daily basis user traffic pressing the like button and messages amassing into high quantities of data that can provide insights to act upon through sources such as analysing web browser cookies and like button presses to facial recognition.

Not only corporations can benefit from analysing data logged. A plethora of academic fields collect data such as mathematics, genetics, biology, physics, sociology. Touching briefly the last field, experiments and observations generate information that are multidimensional and complex to work with in order to simplify them and extract novel patterns and forecast outcomes from conducting computational ethnography and computational linguistics.

It becomes clear that as the computational pipelines incorporate the needs of the market, through generalizing the provided toolset or developing specialized solutions, add yet another abstraction layer on top of increasingly complex systems. Unavoidably, this obscures issues that may rise during application development to harness the power of clusters and distribution e.g an application may slow down due to unoptimized code or a machine running out of memory. Understanding and debugging the execution on distributed systems has always been a challenging problem to undertake due to concurrency, distributed logs and the lack of synchronised clock. For this reason, visualization tools attempt to mediate the underlying state to the user in a digestible way that can assist in detecting bottlenecks and code faults.
1.1 Problem description

During application development on a distributed platform there are a lot of components involved in running the user’s code, such as a variety of data sources, the distributed file system, metadata consistency, synchronization of tasks and failure handling among others. Debugging an application running on a single machine limits the potential sources of error by a large factor compared to the chaotic nature of a distributed environment. The former is easily achieved by using the tools provided by the integrated development environment (IDE) or other tooling like network packet sniffers etc. On the contrary, distributing an application introduces an asynchronous environment, network communication between the nodes to share information, resource allocation and task scheduling to name a few.

Each layer comprising the stack is important as it abstracts the underlying complexity to the layers above. An innate issue of distribution, regardless of how efficient and well designed an abstraction is, is faults of any kind occurring during long periods of time. Thus, being resilient and fault tolerant is an implicit requirement.

The building stone is a distributed filesystem/database such as Google File System(GFS)[4], Hadoop File System(HDFS)[5], which is responsible for storing not only application metadata and files but also be efficient on its network communication. Application submission to the cluster is no trivial task; it means that a central authority needs to be aware of how essential resources are
managed, most important ones being CPU, memory and lately GPU, while at
the same time allocating them in an optimal way, e.g. YARN or Mesos[6].
Finally, a computation engine such as Apache Spark, Apache Beam[7] or Google
Dataflow[8] submit applications to run through the cluster management layer.

To get an overview and insight of the technology stack during the execution,
commercial institutes have developed in house software that works quite well for
their needs. The open source community, and this case Hopswor[9]ks as our
development platform, also has robust and elaborate tools such as Grafana[10],
a generic framework for displaying time-series data from a variety of sources
and Kibana[11], a visualizer for elastic search[12] and the elastic stack[13].

Hopsworks among other functionality(metadata management), offers a web
user interface, supports Spark applications along with Apache Flink[14] and
tensorflow[15] as an abstraction on top of an Apache Hadoop fork, HopsFS[16].
The user has the ability to customize the application’s execution environment e.g.
hardware requirements, application source code, a Kafka[17] data source. As soon
as a Spark application starts, a visualization UI can be accessed displaying in tabs
a variety of tools, Grafana, Kibana, Spark web UI and YARN’s web UI. The open
source community lacks a tool that can summarize the important information and
display them in an intuitive and easy to consume way.

1.2 Problem statement

A Spark application gets executed in a multi tenant environment and as such,
optimizing it requires knowledge of the state coming from each component.
During execution several crucial building blocks may introduce a delay or cause
disruptions that are not easy to figure out without going into logs or reading a
database.

In the open source community there is a need for a Spark monitoring tool
that during runtime, and after the fact, one can use to gain insight and assistance
on pinpointing an issue, e.g. a bottleneck or a machine failure, or margins of
improving application efficiency and cluster resource utilization. Our monitoring
implementation, with Hopsworks/HopsFS as the target platform, aims to answer
the following questions:

- What constitutes a set of potentially crucial cluster information;

- How can these metrics be displayed in a meaningful way?
1.3 Goals

The main goals of the project are:

- Develop a visualization user interface to allow for debugging and optimizing Spark applications;
- Provide useful graphing dashboards that enable the user to correlate the information in order to have an overview of the technology stack’s status.

The above goals can be further broken down into the following sub-tasks:

- Design the dashboards while adjusting the existing software setup accordingly;
- Modify the various components to write their performance metrics to the influxdb;
- Update influxdb schema for the metrics to simplify the query construction;
- Provide a REST API for the frontend to easily query data from the backend;
- Evaluate the impact of the user interface and its performance;

1.4 Reflections of Ethics, Sustainability and Economics

As mentioned earlier, big data collection and processing is present in many fields and environments, which signifies the importance of data analysis as it certain occasions it is favorable to run simpler algorithms with huge amounts of data rather than a specialized approach to exploiting less data. Commercial entities and researchers use it as a way to gain insight and understanding on human behavior, on financial phenomena and extracting genetic patterns.

As mentioned earlier, big data collection and processing is present in many fields and environments, which signifies the importance of data analysis as it certain occasions it is favorable to run simpler algorithms with huge amounts of data rather than a specialized approach to exploiting less data. Commercial entities and researchers use it as a way to gain insight and understanding on human behavior, on financial phenomena and extracting genetic patterns. Executing applications on a cluster means that a large number of machines, physical or virtual, have been allocated. Large corporations and research centers can have their own clusters which are governed by their own rules and regulations. These must include access rights and site protection standards to safeguard the data. The goal of the thesis is to enable the users to tweak their running applications to run more in number and finish the existing ones faster. Same principle applies for
1.5. Structure of this thesis

The thesis document is organized as follows:

- **Chapter 1** gives an overview of the research topic and the problem studied in this thesis. Additionally, it includes a discussion on ethics, sustainability and economics as well as the end goals of this project;

- **Chapter 2** presents the necessary background knowledge to make the rest of the thesis graspable such as a deeper look in Spark and Influxdb;

- The research methods followed are discussed in **Chapter 3**;

- In **Chapter 4**, there is a deeper technical analysis of the solution developed;

- **Chapter 5** provides a discussion on what has been achieved and it what degree the goal has been achieved;

- **Chapter 6** concludes and gives an outlook on the this work and the future pointers;
Chapter 2

Background

This chapter will provide a glimpse into the data storage and analysis tools as perceived and used in the context of this project. Providing a peak into the execution pipeline brings a variety of components onto the table and the interactions between them is complex. Thus, having a good understanding of Apache Spark and its sub-systems, Apache Hadoop ecosystem, Graphite, InfluxDB and the Hops platform among others, is vital for the rest of this work.

These tools are part of the bigger parallel processing movement HPC (High Performance Computing) as depicted in Figure 2.1. Even though supercomputing had its birth 80 years ago with the notion of computers taking up rooms handling military needs, the age of effective data parallel computation started in the beginning of 90s with applications sharing resources such as memory, disk space, CPUs etc. The need for truly durable, fault resilient, automatically balanced load and distributed clusters has been the goal of researchers in academia and industry alike.

One of the most notable attempts was in 2003, when the Google published the Google File System (GFS) paper which was then implemented on Java in Nutch Distributed File System (NDFS). On top of that they built MapReduce that promised to handle parallelization, distribution and fault-tolerance. Both systems, were packaged into Hadoop, under the names HDFS and MapReduce. A quintessential part of the Hadoop framework is Hadoop YARN, that is responsible for managing computing resources in the cluster and using them to schedule user submitted applications. Hops maintains its own open source forks of HDFS and YARN by diverging on many aspects of their vanilla counterpart as they will be described below.
2.1 Hadoop

Apache Hadoop has matured throughout the years and has become one of the most used platforms for distributed storage, big data computation running on top of commodity hardware. The modules comprising the framework have been designed with the strong hypothesis that hardware failures are a common incident and that should be handled automatically. In contrast to HPC, Hadoop has seen a rise in the late years and its market penetration has reached considerable levels making it one of the most established systems to develop on.

The core of Hadoop is composed of a common package abstracting OS scripts to start Hadoop and common utilities, distributed storage file system, HDFS, a resource manager called YARN and the MapReduce large scale computation engine. But the framework’s name has come to represent more than the components above, including many more Apache brand tools, such as Spark, Flink, Zookeeper, Hive, Kafka which are also part of the Hops platform.

One of the pioneering features of Hadoop that came with its distributed nature is that data didn’t move from the machine that retrieved them from the file system. In contrast, the computation code moves to the node, meaning that the computation is done where the data reside. A typical flow consists of a client program submitting the application to the resource manager, which takes all the necessary steps of negotiating with the nodes on the resources to be allocated. Upon completion, the application gets executed in an isolated environment that
2.2 HDFS

Our visualization uses metrics retrieved from Hadoop components, so we need to be able to understand what they mean and extract useful insight from them. In this section, only the necessary knowledge will be presented as the low-level details are not inside the scope of this project. Hadoop follows the semantics and standards of the UNIX filesystem but diverges from it when the performance is at stake such as streaming operations on large files ranging from a few gigabytes to terabytes.

HDFS stores filesystem metadata and files separately. Metadata are stored in a dedicated server called NameNode(NN) and the partitioned files on slave nodes named DataNodes(DN), comprising a fully connected TCP network as shown in Figure 2.3. By partitioning the files, the machine’s capabilities, e.g. disk space, I/O bandwidth, can be extended by adding commodity hardware. To protect file integrity and provide data durability, HDFS replicates data to multiple DNs which has the added advantage of running computation near the required data.

2.2.1 NameNode

A single, metadata dedicated server maintains the HDFS namespace by storing information about files and directories represented as inodes. In a similar fashion to POSIX, records include user permissions, access and modification times,
namespace and disk quotas. Common user operations are served by the NN such as creation, removal, renaming or moving to another directory but other common operations are missing e.g. user quotas. Each operation is written in a file called EditLog and persisted to the local file system, for example renaming or if the replication factor of a file is changed.

By default, a file is broken into blocks of 128MB but this size is configurable on a per-file basis. When breaking a file, the NN needs to maintain a list of block locations to be able to recreate the file in its initial state. HDFS replicates the blocks to different DNs according to user configurable settings and policy. In the case of a DN machine failure then the operation is redirected to another replica.

NN can survive crashes and restarts by saving the entire filesystem and metadata to a file called FsImage, which is also stored in the local file system. On such an event EditLog operations are applied on the FsImage file to get back to the state it was before the crash/restart (checkpoint). At the moment of writing, HDFS does not support multiple NNs making it a single point of failure, rendering the entire HDFS cluster offline.

A vital NN operation is the heartbeat mechanism. All DNs must send a heartbeat message every few seconds, otherwise if NN doesn’t receive one in a specific timeframe it is considered out of service and the blocks hosted on that machine to be unavailable. These blocks will be repositioned unto new DNs. The reply to the heartbeat is used to give instructions to the specific DN. These can be replicating blocks, removing blocks from local replicas, re-registering to the NN and sending an immediate block report or just shutting down the node. Receiving a block reports or a heartbeats from the entire cluster, even if large, is critical to keeping an updated view of the blocks and the filesystem.
2.2.2 DataNode

The DN is the worker of the file system, being responsible for handling the assigned blocks and manipulating their state according to updates received through heartbeat replies from the NN, e.g. creation, deletion and replication. Once it has received a command from the NN, for example read issued by a client, NN sends a list of the DNs having that block which then turns into a direct communication to and from the DN.

The heartbeat mechanism, as presented in 2.2.1, contains information about the blocks stored, as it enables the NN to keep track of the blocks of that DN. Also, it means that the DN is alive and that it can be included in NN’s load balancing and block allocation decisions, since the heartbeat message sends health metrics, total storage capacity, fraction of storage in use and the number of ongoing transfers. The other important message that DN sends is the block report. It is sent when the DN starts up after completing the local file system scanning. The core of the scanning is based on heuristic algorithm that decides on when to create directories and subdirectories to avoid OS restrictions.

To optimally balance and handle the blocks distribution, NN must keep notes on various facets. One of them is to keep track of under-replicated blocks, who are placed in a priority queue with the highest being the blocks in the danger zone of only 1 replica. Another issue that needs to be considered is the block placement, such that a block is protected from machine failures but also offer blocks efficiently to the clients touching the thin line of data reliability versus write performance. For example, the default block placement policy of HDFS is to write a block where the writer is located, with the other two replicas placed in different nodes on a different rack.

2.3 YARN

Hadoop 2, broke the tight coupling between the computation and storage layer by introducing a middleware, the resource manager called YARN[1]. This opened the Hadoop platform for use by other data computation frameworks, as up to that point was only available to MapReduce. This work was the burden of the MapReduce’s module, the JobTracker, responsible for managing the resources in the cluster, by tracking live nodes, the map and reduce slots as well as the coordination of all tasks running in the cluster, restarting failed/slow tasks and much more. Scalability was an issue with a single process having so many responsibilities, especially on big clusters.

On the other hand, YARN abstracted the resource managing and distribution of processes. There are three main components in YARN, the Resource Manager(RM),
Figure 2.4: YARN RM and NM interactions

the Node Manager(NM) and the Application Master(AM). The master daemon, the RM, communicates with the client, tracks cluster resources and manages work by assigning tasks to the NM worker nodes. In the moment of writing the RM can schedule two resources, vcores and memory, with GPU support coming next. VCore is an abstraction of the physical available CPUs of the machine, used to make CPU scheduling possible and signifies the time share of a physical core. When an application is submitted the AM is responsible for allocating the needed resources for the application to run on. As such a framework that needs to run on the cluster needs to use its own implementation of the AM.

2.3.1 Resource Manager

The RM, a single entity in the cluster as mentioned above, aware of where the worker nodes are located and what each can provide in terms of resources which forms a global view of the cluster, allowing for optimal cluster utilization including constraints such as fairness, capacity guarantees (keep resources in use all the time) and honoring the client agreements or the SLA. RM also supports the replacement of the built-in scheduler with a user defined one, in the case of the need to use different fair and capacity scheduling algorithm.

To satisfy its role as a coordinator the RM is composed of a collection of modules. The clients can submit or terminate an application, obtain a variety of metrics through an RPC interface listening on the RM. This request type has lower
priority than the admin service that serves cluster management queries through a different endpoint to avoid being starved by the client’s requests. The list below concerns only

The next sub section of services is the one to talk to the cluster nodes. Working in a volatile environment means that nodes restart, crash or get removed from the system. Each node must send a heartbeat message on an interval to the RM's NMLivenessMonitor service so the RM can keep track of alive nodes. By default, if a node doesn’t communicate with the RM in 10 minutes it is considered dead and all the containers running on that machine are removed and no new ones are scheduled on that node. The second tracker service is called the NodesListManager that keeps track of the valid and excluded nodes. Both services work tightly with the ResourceServiceTracker that exposes RPC endpoints that can register new ones, blacklist dead ones, respond to heart beats as well as forwarding them to the YarnScheduler.

On a similar page, are the services provided to serve AM requests. The ApplicationMasterService handles registrations and termination of new AMs, to retrieve information about container allocation and release and forwards requests to the YarnScheduler. The AMLivelinessMonitor serves the same purpose as the NMLivenessMonitor, with containers that are allocated to that application getting marked as dead and the RM can attempt to re-schedule the AM 4 times by default.

At the heart of the RM is the scheduler and its related modules. The user exposed interfaces are protected by the ApplicationACLsManager by maintaining an ACL list and every request must go through that before executing a command on the cluster. An ApplicationManager is responsible for storing and updated a collection of submitted applications and caches completed applications to be served via the web or command line user interfaces. New applications submitted
via the above API need an AM and as such an ApplicationMasterLauncher is responsible for keeping a list of new or re-submitted failed AMs and cleaning up when an AM has been terminated or completed execution. The YarnScheduler is responsible for allocating resources, as described in the first paragraph, to running applications by adhering to the application SLA, queues, available resources etc. The final module, ContainerAllocationExpire, makes sure that the allocated containers are actually used by the AMs and then launched on the respecting NMs. The plugin system exposes the cluster to unsigned user developed AMs, potentially malicious, so the module verifies if a container hosting NM reports to the AM, where in the negative scenario the containers are killed.

2.3.2 Node Manager

There have been several mentions of the NM in the preceding sections but in this one its functionality will be summarized in a concentrated way. The NM is the per node agent that is responsible for managing the container’s resources, namely memory and virtual CPU, communicating with the RM, monitoring the health of the node itself, log management and file keeping as well as running utility services for various YARN applications.

For a NM to bootstrap itself, it runs a NodeStatusUpdater who registers the node to the RM and sends subsequent calls for container status updates, e.g. new containers instantiated, completed, crashed containers or wait for a kill a container signal from the RM. At the core of the NM is the ContainerManager, who incorporates a collection of sub-modules. An RPC server is waiting for AM requests to create new container or terminate the execution of one. In any case, a ContainersLauncher service is ready to spawn a thread from a threadpool to complete the task as quickly as possible, where in the case of launching it expects detailed specifications as well as information about the container’s runtime. After the fact of launching a new container, the ContainersMonitor looks over the launched containers’ resource utilization on the node. In the case that it exceeds the set limits the container is killed to prevent stray containers from depriving the rest well-behaved containers running on the NM. The final piece comes into the form of an extensible utility functionality support which allows for framework specific optimizations such as MapReduce shuffle operation.

2.3.3 Application Master

The entity that oversees the job execution is the AM, spawned by the RM’s service ApplicationMasterLauncher. Through the client it will receive all the necessary
information and resources about the job tasked to monitor. It is launched in a container that will most likely share physical resources with the other containers, which causes several issues, one of them being not being able to bind on a pre-configured port to listen on. When the job launches the client-program will receive a tracking url that it can use to keep track/job history for the job. On an interval, the AM sends a heartbeat to the RM to notify that is still up and running.

When the client submits a job, the AM must send a request to the RM containing the number of containers required to run the application along with the necessary specification such as memory, CPU and the node’s location to which the RM replies with an AllocateResponse. Upon receival, for each container sets up the launch context which is fed the container id, local resources required by the executable and its environment, what commands to execute to name a few. When this process is completed the container is initialized and launched. Upon completion, the AM unregisters itself from the RM as well as from the NM it has been running on.

2.4 Data computation

2.4.1 MapReduce

One of the essential components of the Apache Hadoop 1 is MapReduce[22]. But before delving into its details and insides, it’s important to know the trigger of its existence. Around 2003, Google was processing a high amount of data in their datacenters but the existing file system at the time suffered from a few issues. It required for the form of data had to be based on a schema, it was not durable, the capabilities to handle component failures were lacking such as CPU, memory, disk, network and last that the load was not re-balanced automatically. The solution came into the form of the GFS paper[4].

Having NDFS in place, which later evolved, and merged with the Apache Hadoop project, Google engineers needed a framework to harness the benefits of a distributed storage engine and create a uniform cluster computation engine that will replace all the custom tools that have been used in the company so far. The end product of this was the MapReduce framework, as a goal to simplify the data processing process on large clusters by abstracting the complexity and intricacies of a distributed environment. It provided very appealing properties out of the box – parallelization of tasks, distribution of work and fault-tolerance. The need for a distributed solution got satisfied, initially, and companies such as Facebook, Twitter and Linkedin started using Hadoop in their clusters.

The fundamental primitives, without surprise, are map and reduce finding their
stems from functional programming paradigms. Functional languages support parallelism innately as functions and operators don’t modify a data structure, only create new, modified ones. This sets a demarcation line between the initial data and the output since more than one threads can use the same source but apply different transformations without affecting the others. By having the ability to pass functions as arguments the developer can pass the output of one MapReduce operation to the input of another.

By using the map operator, the same function is applied on each chunk of data, outputting potentially a new data type, not related to the input one. After applying the map operation, the transformed chunks need to be merged and processed, or reduced. While map accepts X items and outputs the same number of items, reduce takes X items and outputs a single one. It takes an accumulator and it runs on the first element, adds it in the accumulator and then its fed the next and so on.

Following these principles, the Google engineers built a framework that mimics this behavior by modelling the problem at hand by using Mappers and Reducers. As seen in Figure, the user writes the two functions using the MapReduce library. Each chunk of data is fed into a mapper and produces a set of intermediate key, value pairs. The library under the hood, aggregates these results by key and passes them into the reducers. The reduce function receives a key and a list of values, potentially creating a smaller set. The drawback of this design is that until all mappers have finished processing, the reduce phase cannot start.

The hello-world example is the famous word count. A set of documents is parsed into key, value pairs where the key is the document name and the value its contents. Each mapper receives a document and for each word outputs ['word', 1], meaning that the word has appeared once. Once all mappers have finished processing the documents the intermediate phase transforms the words into keys...
and the value is a list of 1s. Then the reducers take that and sums up the numbers and emits it, e.g. ['to', 10]. Upon completion of the program all words have been reduced into words and their count across all documents.

Optimality in MapReduce is based on several assumptions that need to hold true. The volume of data must be large enough such that breaking it down into smaller chunks doesn’t affect the overall performance. The computation can be problematic if there is a requirement on using external resources including other mappers. The sheer size of input data is expected to be reduced by the end of the process in such a way that the goal aggregation has been satisfied. Above all though, is the master coordinating node, which must be robust, responsible and fully functional throughout the execution.

Since its release, MapReduce paradigm has been developed and forked to cover the needs of each. In this section, we will go through the vanilla version as it was first introduced by Google in 2004. As seen in Figure 2.6, initially the client program, depending on the underlying storage framework, using the framework library splits up the data into 64MB usually as it is the one used by GFS. One of the machines in the cluster is assigned the role of the coordinator called the master while the rest of them are the worker nodes that hold a slot. This slot can be either map or reduce but not both. The distribution of tasks attempts to minimize data movement pairing the map with data that run on the same physical location.

A worker node that has been assigned the map task reads a slice of data that are parsed into key value pairs and passed into the mapper program. The intermediate results are temporarily written into a buffer that when it becomes full, it is partitioned into regions and written to the local disks. The master gets notified of the location of these regions that are later passed to the reducer nodes. As soon as the reducer is notified of these locations, it reads the data from the map worker disks through an RPC operation. The keys are then sorted to be grouped together as a key can occur more than once across the data. Subsequently, these are given to the reducer function that iterates over the values of a key and applies the user defined reduce function. Each reducer has its own output file that is named according to the user’s specifications. From this point, these files can be passed into another MapReduce function, a different framework that can deal with partitioned data or just merged into one file.

To coordinate the process, master node maintains the state of each map or reduce task which can be idle, in progress or completed. Along with this list, the machine stores the identity of each worker node and the location and sizes of the completed tasks’ intermediate file regions. As each task is completed the location list gets updated. To survive failure, the master node periodically writes a checkpoint and when it crashes it just restarts from the last checkpoint.

Coordinating implies that the master keeps tabs on the worker’s health status. Each worker periodically sends a heartbeat to the master. In absence of heartbeat,
the master tags the node as unresponsive and all the map tasks that have been completed are rescheduled for execution by other workers. In a similar fashion, map or reduce tasks in progress will get rescheduled by getting set to idle. The completed tasks must be rescheduled since the local disks might be inaccessible on the unresponsive node. When a task gets rescheduled, the master broadcasts the event so that reduce workers can find the expected results in the correct locations.

2.4.2 Limitations and criticism

MapReduce was a pioneering feat for its time but it had its drawbacks and limitations in several aspects[24]. It was praised for its throughput in processing big amounts of data, fault tolerance and able to scale horizontally, not so for its processing time as it would fluctuate in the hours. As it was using HDFS for the shuffle phases, a lot of time was spent on serializing, IO stalls and block replication. The process of bootstrapping an application was expensive, so in the case the input data were small enough, the processing would be better if it ran on a single machine by simply executing reduce(map(data)).

Only supporting map and reduce operations meant that a potential application should be designed and implemented in a way that would fit the model. This required deep and extensive knowledge of the developed system architecture as even fairly simple operations such as joins on data sets were not trivial to realize. Complex iterative chains were realized by chaining MapReduce jobs with the output of one serving as input for the next. Algorithms that shared state between phases were not suitable such as machine learning or graph processing. As a solution to this problem, HDFS was used to store intermediate results which induced times that are several orders of magnitude higher than running a query in a modern DBMS system.

Last point of criticism was the complex and sensitive configuration parameters, file block size, number of parallel tasks for example, that demanded knowledge of not only the characteristics of the workload but also of the hardware. Not optimally tuned job resulted in under-performing execution and under-utilized cluster resources.

2.4.3 Spark

Running into the limitations of the MapReduce cluster computing paradigm, researchers in University of California’s AMPLab brought Spark[25] to life, which later became part of the Apache ecosystem. Apache Spark is a distributed general-purpose cluster computing framework that offers a programming interface
for running applications with implicit data-parallelism, fault-tolerance and scalability, available in Java, Scala, Python, R and SQL. Its designers have broken down the architecture into the core and on top of it, modules that provide machine learning[26], ETL, graph processing and analytics capabilities.(Figure 2.7).

Spark ecosystem contains a family of closely interconnected components to offer a platform to execute a variety of applications and interoperating tightly, combining different processing models. At the heart of the computational engine, is the core exposing a unified interface to the modules built on top of it. A benefit for offering such a tight integration is that when the core engine adds an optimization, all modules above will benefit from it. Additionally, the complexity of maintaining a lot of software tools is minimized, as the developers only need to maintain one tool. Having a wide array of services, Spark is used by data scientists for data exploration and experimentation, software engineers to run machine learning and much more.

The processing core includes the basic functionality, including fault tolerance and recovery, interacting with the storage systems through a plugin system, memory management, task scheduling for instance. The cornerstone of Spark’s performance is the immutable, in-memory data structure called Resilient Distributed Dataset(RDD)[27]. They represent a collection of items distributed across the network and can be manipulated in parallel, all through a programming interface to manage and control them.

The modules, as mentioned above, cover a wide range of computational models. Spark SQL[28] is a package for working with structured data and allows for querying them via SQL or the Hive variant, Hive Query Language. Sources include Hive tables, json, parquet, Avro, jdbc and can be manipulated in a unified data access pattern, it is transparent from the programmer’s point of view. On top of SQL, Spark also comes with a machine learning library that includes multiple types of ML algorithms e.g. regression, clustering, classification. GraphX[29] is a graph manipulation library, transforming the RDD interface into one that can run graph algorithms on nodes and edges with arbitrary properties.

Being a batch processing engine, Spark in version 2 introduced the ability to stream data through the Streaming package. This continuous stream of data is represented a[30], a sequence of RDDs. Live environments suffer from several issues, record consistency, fault tolerance and out of order data. Spark Streaming tackles these issues by making a strong guarantee that at any time the output of the system will be equivalent to running a batch job on a prefix of data. The data are treated as if they were new rows into an input table. The developer defines a query on the data as if it was a static table, since Spark automatically converts the query resembling batch into a streaming logical execution plan. The output of a streaming operation, based on the output policy (append, complete or update) enables the developer to write the changes into an external storage system, such
Figure 2.7: Spark platform components [31]

as HDFS, Kafka, database.

As an example, let’s go over how a developer would design a word count application from a Twitter feed in the streaming model. To run this query incrementally, Spark must maintain the word count from the records seen so far, update accordingly as they arrive in micro-batches and pass it through to the next micro batch arrival. The new data are fed to the query and the results are written to the output based on the output policy, in this case update the word count so far.

Interactive and Iterative jobs require fast data sharing and MapReduce suffered due to its swap data being written on HDFS between iterations, resulting into spending most of the time in IO, replication and serialization operations. On the other hand, RDDs may get created on the driver by running parallelize, or on executors that load data from HDFS once and then saving the intermediate results on memory, which raises memory in to a valuable computational resource. The latency, on the other hand, of such jobs may be reduced potentially up to several orders of magnitude.

One can apply two types of operations on an RDD: transformations and actions, transformations create one or more RDDs. An example of this is a map operation, it accepts as an input each dataset and after passing it through a function it returns a new dataset. On the contrary, actions take as an input a collection of RDDs, runs them through a function and returns the result to the driver program e.g. the reduce function. These operations are lazy, in the sense that results are not computed immediately, the runtime environment remembers the transformations to be executed when there’s an action call in the program. This allows for Spark to run tasks more efficiently as the driver knows a-priori the application’s logical execution. As a use case, the program might have initiated a map operation but the next operation is reduce which means that an executor will return only the last result instead of a huge dataset.
A technique that gives RDDs the resilience property is the lineage or the dependency graph. It’s a way of maintaining the parents of an RDD which allows for re-computing a partition in case it’s corrupted or lost. When applying transformations to an RDD, Spark does not execute them immediately. Instead it outputs a logical execution plan. These graph nodes are the result of the transformation and the edge is the operation itself. As we can see in Figure 2.8, the earliest ones have no dependencies or refer to cached data and the bottom nodes are the result of the action that has been called to execute. In the case of a simple application, such as loading a file, filtering the contents and then counting them, the runtime builds a graph with a node for each distinct operation. But due to the lazy execution, the logical execution plan will only be realized and executed when the count action is called to return a result to the user.

The Spark core is designed to efficiently scale horizontally, from a single home computer to thousands of nodes on a cluster. To manage it and at the same time maximizing flexibility, an application can run on a cluster mode, instead of a local one, using variety of cluster managers, for instance on a fresh installation one can use the Standalone cluster manager that comes out of the box; if available Spark can run on top of YARN(see section 2.3) or Apache Mesos[6] cluster managers.

The application to run in a distributed mode, it must be broken down into smaller tasks. The coordinator program is called the driver, the driver node called a master and the worker programs executors. A driver is a JVM process that hosts the SparkContext for an application, responsible for connecting to the chosen cluster manager, which allocate application resources across the machines. Once a running environment is created on a machine, Spark acquires an executor on each, that will run computations and store application data. Being data-centric, the next step is to send the code to be run (JAR, or python files) on an executor that executes tasks delivered by the SparkContext(see Figure 2.9).

The driver entity that coordinates the computation work around the cluster is called the DAGScheduler. It converts a logical execution plan to a physical one using stages. Upon an action’s call, the context provides the DAGScheduler with the logical plan to transform it to a set of stages(explained below) and submitted as
sets of tasks for computation. The core concepts of the scheduler are the jobs and the stages that are being tracked through internal registries and counters. A job is the top level computation item in response to an action’s call. This boils down to acting on the partitions of an RDD that the action has been called upon. A job breaks down the computation into stages, that in turn contain a set of parallel tasks that have one on one relation to the target RDD’s partitions.

The last function that needs to be presented is the shuffle. DAGScheduler builds up the stages according to the RDD’s shuffle dependency. Shuffle means that the data have to be moved, either locally or through the network to the next stage’s tasks, for example from a stage working on an RDD that goes through a map transformation fed to the next stage’s reduceByKey.

### 2.5 Telegraf

Collectd[33] is a stable and robust tool but cumbersome to work with. The InfluxDB developers have released as part of their time series platform, called TICK[1], a data collection agent that runs on each host machine, collecting, processing and aggregating metrics from the system, third party software running on it (databases, message queues, networking etc) on an interval. This information gets written on InfluxDB that we use for running aggregation queries with out of the box functionality. Telegraf’s[34] configuration is fairly simple, allowing for easy configuration of in-house and third party plugins, one of which is the one that outputs the metrics to InfluxDB. Due to the agent’s long timespan, the developers have designed it to impact the machine’s memory as little as possible with several
2.6 Graphite

Having covered the distributed data storage and computation frameworks, the next logical step is the collection of the software’s metrics to visualize them. Graphite[35] is an open source tool written in python that collects, stores and displays metrics. We are concerned with the first two parts, as the last one is basic for our needs. It is not a data collection tool, rather than a way of storing and displaying information but it provides many integrations, making it a simple process to write application data.

The first module is called carbon[36] as shown in Figure 2.10, which is a Python Twisted daemon that listens for time-series data on an open socket port and graphite IP address. It exposes a REST interface for easy programming integration, if there’s not already support for the used tool. Depending on the requirements, the developer can use the most straightforward way of sending data by using the plaintext protocol[37], e.g. <metric path> <metric value> <metric timestamp>. The metric path is the variable name is in the form of dot delimited components. In case of large data, using the pickle protocol is preferred since they can be batched and compressed, and then sent through the network socket. If the above choices are not sufficient, the last way of sending data is by sending messages as defined in the AMQP standard[38].

By default, Graphite uses whisper[39] to store, UNIX epoch, time indexed, information on disk. It is a fixed-size database, meaning that once its full, the latest data are written on top of the oldest ones. A whisper database contains one or more ordered archives, each with its own time resolution and retention policies, from highest-resolution and shortest retention to lowest and longer respectively. For instance, when a data point arrives it is stored in all possible archives at the same time, aggregated if needed to fit lower resolutions. Being a very simplistic database, it might introduce bottlenecks when the incoming queue increases under heavy load.

The last piece comes in the form of a web user interface built on top of Django[40]. It provides an easy and simple way to expose the metrics stored on the database by using one or more graphs. The user can create, delete, merge and customize each graph and create/save dashboards for future use such as operations monitoring. Even if this UI will not be used in this project, it brings up a useful feature, applying wildcards on metrics to filter and aggregate them. For example, applying the template filter server.*.cpu will match all the metrics coming in from different servers, say server.dn0.cpu and server.dn1.cpu.
2.7 InfluxDB

Luckily, Graphite allows for replacing the default database with one of the other available options. InfluxDB[41] is the database that is already integrated into Hops. InfluxDB stores data of the same nature as Whisper but on a more robust, performant and scalable implementation. As expected, from the InfluxDB side a plugin needs to be set up to listen to Graphite’s carbon plaintext protocol. Being built with this specific task in mind, it offers an SQL like query language that is enhanced with built-in time-specific functions, such as mean, max, derivative but also by default exposes a REST interface to interact by sending a simple HTTP request.

Instead of using tables, InfluxDB stores data points into measurements. Being a time series database, a time column is included by default in each measurement, an RFC3339[42] formatted timestamp. Each column can be either a tag or a field. Tagging a column, is an optional operation, that marks it as an index, leading to faster times when using them as metadata filters. On the other hand, field is a required column to have data in the database. In case these are used as filters then the query must scan all the values that match the condition, rendering it sub-optimal. A measurement can also be part of different retention policies, by default is the autogen, governing over the lifetime of data in the database and what the replication factor should be, in case of cluster mode. Every possible combination of tags, retention policy and measurement name comprise a series which is essential when designing the database schema, as when series number goes to high, memory usage will increase.

The built-in query language, InfluxQL[43], supports a wide range of operations.
The query language has implemented only a basic subset of the SQL query structure such as select, where, group by, order by but the ones left out are inner and outer joins which would be welcome in time of development. As a temporary replacement subqueries can be used. The strongest tool of InfluxQL are the built in mathematical and aggregation functions that work seamlessly together with the group by clause, e.g. derivative, integral, median, stddev, bottom, max, min, percentile and much more.

InfluxData\(^{[44]}\), the maintainer of the platform, offers a suite of open source tools that comprise the TICK stack. The first letter in the acronym, Telegraf\(^{[34]}\), an agent written in Go that collects, processes, aggregates and writes data to a wide array of options, one of them being InfluxDB. Out of the box, the user can retrieve data from systems, services or third-party interfaces. In this project, it is used to collect OS level metrics.

2.8 Related work

Before moving to the implementation details, it is essential to review the current solutions in the market and what is our motivation for going forward with this work. The tools that emerged from the data computational era are numerous and they attempt to reimagine the way users gain insight and useful information for tuning, optimizing and monitoring applications and resource allocation, whether that be an open source or proprietary player.

The open source community can offer a wide selection of visualization tools, aka dashboards comprising a collection of services such as graphs, widgets, labels, notifications and alerts. Dr.Elephant\(^{[45]}\) is a monitoring tool developed by LinkedIn with a goal to optimize cluster resource allocation and as such improving the efficiency of, not only, Spark jobs. On an interval, Dr.Elephant retrieves applications that have completed successfully or terminated from YARN\(^{[1]}\) and processes them with a set of heuristics and comparing the results with the history of each execution of the application.

Kibana\(^{[11]}\) covers the visualization part of the, as advertised, Elastic Stack\(^{[13]}\) which comprises a data collection, search, analysis (machine learning, graph exploration among others) and visualization technology stack. Being part of a tightly closed ecosystem, it only visualizes information coming from Elasticsearch\(^{[12]}\) clusters, such as log files, geographical data and metrics. There are a few, dormant, open source projects that report Spark metrics to Elasticsearch but that entails that this project will be limited to the Elastic Stack or get weighed down by the additional complexity of pulling data from an additional source.

Spark out of the box provides a web user interface that gets exposed from the driver and it is accessible from the port 4040, while the application is running, or
Table 2.1: Comparison of Spark monitoring tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Live updates</th>
<th>Extensible &amp; ease</th>
<th>Offline</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark web UI</td>
<td>Refresh page</td>
<td>N/A</td>
<td>Yes</td>
<td>Stable 4.3</td>
</tr>
<tr>
<td>Grafana</td>
<td>Yes</td>
<td>Have to develop plugin</td>
<td>Depends on data source</td>
<td>5.4</td>
</tr>
<tr>
<td>Kibana</td>
<td>Yes</td>
<td>Ongoing API changes</td>
<td>Heuristics</td>
<td>Stable 2.0.6</td>
</tr>
<tr>
<td>Dr.Elephant</td>
<td>No</td>
<td>Simple process UI not extensible</td>
<td>Code only</td>
<td>Only offline, parses logs</td>
</tr>
<tr>
<td>Vizops</td>
<td>Yes</td>
<td>Fair bit of work UI fully extensible</td>
<td>InfluxDB/REST</td>
<td>Alpha</td>
</tr>
</tbody>
</table>

through the history server for viewing after the fact through logs that get cleaned up after their lifetime has ended. Its focus is not displaying information in an easy to digest or elegant way, save a few graphs, but offering deep and detailed information in a tabular form about an offline snapshot of an application and its sub-components. Through the tabbed panels, one can view metrics about the jobs, stages, tasks, storage (RDD[27] and memory usage), executors and, if applicable, streaming and SQL.

One of the most popular visualization tools among the open source community is Grafana[10], offering editable graphs, customizable dashboards, plugin support and metrics from a variety of data sources, including Graphite and InfluxDB. Having such an active community, one can build dashboards with panels, that range from zoomable graphs (line, pie charts, histograms etc.), to tables and clocks filtered by a time range panel. For each panel, the user can set up the data source and the query to run on, the point granularity (e.g. InfluxDB group by), title, column span, legends, alert etc. Offering such an infrastructure brings along complexity with little control over how the core code works to add or experiment with new visualizations, ending up losing ease of use from both developer and end user side. It has been built on a javascript library called FlotJS[46] that hasn’t been updated for 3 years, at the moment of writing, and its documentation is lacking, even though there are attempts of introducing D3.js[47] panels[48].
Chapter 3

Methods

The visualization’s goal is to provide information about a Spark application and the related multi-tenant environment that will ultimately help in optimizing the application and its resource usage across the cluster. The static nature of the displayed source of information pushes the focus onto picking crucial information, that will be split across our visualizations, on a qualitative basis. To develop such a network of graphs that will interact, so to say, requires extensive reviewing of the current Spark visualizations in the market and the metrics that we have access to. To verify the impact of this work, that will be built up in the next two chapters, we present several scenarios that the graphs assist in identifying issues in the application or pinpointing optimization margins by evaluating the experimental data (CPU and memory) from a quantitative, as well as qualitative standpoint.

Graphite and InfluxDB are responsible for collecting the data. This sets the demarcation line of an iterative and deductive process on finding out how a core set of information that will cover the state of the monitored components, namely Spark, YARN and the hosting machines. Having the data is not enough as they need to be processed and displayed in such a setup that will be meaningful for the end user. Meaningful in a way that will be easier to get an overview than, the main frame of comparison, the Spark web interface UI, as presented in Related works (section 2.8), but also use the latter in conjunction with this work.

The experiments were conducted in a cluster of three virtual machines, each with 15GBs of memory and 2 virtual cores and located on different host machines to verify our research questions. In a real world scenario the experiments would be conducted on a larger cluster, if not the thesis time limitations. For the quantitative analysis we chose to execute a CPU intensive application, specifically approximating digits of Pi. The reason for this is that it allows for a predictable experiment, being a solely computational problem, yet sufficient to present the tuning process in various resource configurations. The other class of issues is that of qualitative nature e.g. skewed input, cluster machine crash etc, and as such
we inspect the memory usage of the Pi approximation application and shutdown one of the three available YARN NMs. After verifying that our questions are indeed answered, we conclude this work by generalizing the effectiveness of the visualization in a larger scale as a distributed application would follow similar patterns as in our experiments.

3.1 Data collection & analysis

The true value of data comes, in the project’s context, from helping the user identify what slows down an application, how resources are being utilized and, in general, truly understanding the dynamics between components of the technology stack. Each one of these building blocks, as explored in Chapter 2, write their internal state information into InfluxDB through either Graphite or Telegraf via HTTP requests.

Our tool’s goal is to provide the user with information to determine the bottlenecks of the application, whether these are CPU, network, memory and/or disk/IO bound. Understanding what is happening on each layer of the stack can save development time and assist in reaching the optimal cluster resource allocation. To make this requirement concrete, this project must provide sufficient instrumentation and monitoring via visualization aids to achieve the aforementioned goal.

Running a Spark application involves many agents, fact that fills the table with a lot of facets to keep track of during execution. Since the applications run through a cluster manager we need to be aware of the state of these machines and the provisioned resources e.g. containers to run tasks on. In Hopsworks’s setup, an application depends on YARN for resource managing, HDFS and local disks for file storage, Kafka[17] for message digestion from a streaming environment, NDB[49] to store crucial information related to authorization, authentication, project management etc. and finally InfluxDB for metrics storage.

The following are the areas that the visualization will attempt to assist with during runtime or after completion:

- **Storage systems**: an application will have to read some input data from an HDFS cluster. The simplest way to run these together would be by running Spark standalone mode on the HDFS machines. But this is not easy to do, so the way is to run both frameworks through YARN. Tuning them to harmony depends on a few environmental variables, the machine’s resources and much more. Low performance or not fine tuned application may slow down due to IO caused degradation;

- **Local disks**: When partitioned data doesn’t fit in ram or memory to preserve
shuffle data between stages, Spark spills them to the local disks, which might be the same disks as HDFS. Keeping an eye on the spill size is crucial and depends on the data partition size, number of executors, physical disk IO capabilities, network congestion etc;

- **Memory**: the cornerstone of Spark’s performance as it minimizes reading from disk, an application can run with as little as 8GB up to hundreds of gigabytes of memory in each machine. It is very common for an application to crash due to an out of memory exception whilst running very expensive task(s). Additionally, there should be space for the OS/JVM/buffer cache to function properly;

- **Network**: Even if the application uses only memory, it can still slow down due to network low speeds, especially when executing reduce, groupBy, any other aggregating function that causes cluster wide traffic or that the host machine is under heavy load;

- **CPU Cores**: CPU is responsible for the computational heavy lifting, decompressing serialized data from storage and making the application CPU bound is desirable for both batch and streaming environments;

It might be getting repetitive but Spark gorges on memory. Out of memory exceptions are the most common errors occurring during an execution, are thrown due to the driver or one of the executors running out of JVM heap space. An application can run out of heap memory from issues caused by the level of parallelism being too low i.e. tasks are too few and they read too big files relative to available memory, an aggregation function during the shuffle stage e.g. reduceByKey, groupByKey, memory leak in the user code, unsuitable serializer or just the YARN driver/executor containers not having enough memory.

The JVM during execution needs to identify and remove unused objects from the heap space by using a mechanism called *Garbage collection* [GC] [50]. Spark during an application execution will potentially create a large number of volatile objects, thus keeping heap memory as clean and available as possible, is a quintessential performance tuning parameter. While GC is running, the application does not make any progress since the JVM halts all activity. On top of that, if GC executes for long durations or being too frequent, it is unavoidable that the application will slow down or crash [51] [52].

Even allocating and managing the available memory efficiently, the properties of the input data can cause slowdowns. Imagine a Spark streaming application reading a stream of US located Twitter posts during the night of a famous baseball match to count the hashtags used. As expected a large portion of them will be related to the match, leading to a skewed input set or RDD partitions e.g. Partition
1 has X items, 2 has 10 times more etc. During the execution of aggregation by key functions a small portion of tasks will be running for far longer than the rest since they will have to process inversely proportional number of keys. This results to an application waiting for a few tasks to, potentially never, completing their execution.

The health and performance of the physical machines, that comprise the cluster, is equally crucial with the application’s qualitative and quantitative qualities. Shuffle data is transferred through the network, and in the case the machine is overloaded, an application may face slowdowns and overall performance degradation. With modern hardware, network speeds raise the threshold high enough that an application is more likely to starve from other resources, such as virtual cores, IO speeds, disk space or even the state of the machine itself might be far from optimal.

On top of the above, the streaming functionality introduces new capabilities and issues to monitor[53]. A special family of tasks, called receivers, are running throughout the application’s lifetime with their job being to read data from the input sources, storing them into memory for further processing and replying with an acknowledgement if the source supports it[30]. As streaming applications will have to be running for long periods of time, a streaming job’s performance is vulnerable to any cluster failures. The driver places an incoming batch in the queue for further processing which can be significant compared to the actual processing delay. In case the processing rate drops below the input rate, back-pressure is being built up across long periods of time resulting to input lag. Lag is not only visible inside a Spark application’s pipeline, but also in the streaming source for example a Kafka consumer lagging behind a Kafka producer in terms of message consumption[54].
Chapter 4

Implementation

With the background knowledge established, the next logical step is to break down the project’s requirements. The end goal is to implement a graphical interface to overview the status of the running or completed Spark application along with the other components that comprise this multi-tenant environment. This chapter is broken down into subsections, each one covering a building block in the technology stack:

1. Metrics collection from the various sources;
2. Choosing a database schema;
3. Retrieving metrics from the backend through a REST API;
4. Picking a graphing library to display the information;
5. Gaining insight from the data;
6. Dashboard implementation;
7. Overview of the process of adding a new graph in the dashboard;

4.1 Collecting metrics

The fundamental block for this project is for the monitored components in the Hops technology stack to report their status and performance metrics. Each framework provides several options for exposing their internal state and the most popular ones are via a REST API, a Graphite plugin (see section 2.6) or JMX beans, which boils down to a REST endpoint request. First decision point comes on if the data retrieval model is a pull or a push. Pulling data means that there
must be an agent running that will poll an endpoint during application execution. This approach adds complexity to the system since we must maintain state. On a similar fashion, the JMX retrieval is also on a similar complexity level as the REST approach. All three components that are under monitoring provide, through their respective settings, a way to send the metrics through the Graphite listener and subsequently to InfluxDB. For monitoring the host machines, a plugin agent called Telegraf was used, developed by the InfluxDB team, which will be analyzed below.

Both Spark and Hadoop 2, use a file for setting up the data collection, for example the source and the target of the data. In the case of Hadoop, the same file is used for both YARN and HDFS as each of their subcomponent is considered just another data source. The commonality between the metrics files above is that for the source the developer just points to the java class that is responsible for aggregating the metrics. As far as the sinks are concerned, the relevant setting must point to the Graphite sink class and server. There are two more settings that are crucial, the graphite prefix of the data and the write metrics interval. The former is important as this is what will be used to transform the data to a table in InfluxDB and the latter must be chosen to balance the granularity of the graphs and the size of the written data.

Spark’s metric system is set up using a file called metrics.properties. Each source instance needs to be set up to send its metrics to Graphite but Spark’s
4.1. COLLECTING METRICS

Listing 4.1: Spark metrics settings

```java
*.sink.graphite.class=
    org.apache.spark.metrics.sink.GraphiteSink
*.sink.graphite.host=192.168.56.101
*.sink.graphite.port=2003
*.sink.graphite.period=1
*.sink.graphite.prefix=spark

# JvmSource for master, worker, driver and executor
master.source.jvm.class=
    org.apache.spark.metrics.source.JvmSource
worker.source.jvm.class=
    org.apache.spark.metrics.source.JvmSource
driver.source.jvm.class=
    org.apache.spark.metrics.source.JvmSource
executor.source.jvm.class=
    org.apache.spark.metrics.source.JvmSource
```

developers have allowed for the use of wildcard to apply the same property for each source. A source can be the master node which is the Standalone cluster manager coordinator or its workers, the driver or all executors. In Hops, we use YARN as the cluster manager and as such metrics from the master and worker nodes are not going to be used, but included in the properties file for completeness. Automatically, the metrics class from each source is registered even though we explicitly enable the JvmSource only. For example, by setting the source of each executor to JvmSource, Spark also adds the ExecutorSource in the metrics registry. The data from the sources need to be written to a sink. Spark by default provides a wide range of options, such as exposing Java beans through JMX, writing to a CSV file, Ganglia, a metrics servlet for REST consumption or, our chosen, Graphite sink. Setting up the sinks is trivial, as they only need the information described above.

```
spark-submit <other parameters>
--files metrics.properties
--conf spark.metrics.conf=metrics.properties
```

The data from the sources need to be written to a sink. Spark by default provides a wide range of options, such as exposing Java beans through JMX, writing to a CSV file, Ganglia, a metrics servlet for REST consumption or, the
Listing 4.2: Excerpt from Hadoop nodemanager settings

```
*.sink.graphite.class=org.apache.hadoop.metrics2.sink.GraphiteSink
# default sampling period, in seconds
*.period=2

nodemanager.sink.graphite.server_host=192.168.56.101
nodemanager.sink.graphite.server_port=2003
nodemanager.sink.graphite.metrics_prefix=nodemanager
```

chosen source, a Graphite sink. Setting up the sinks is trivial, as they only need the information described above. With the properties set up, it’s enough to start seeing metrics in InfluxDB, using the Graphite plugin. Special attention needs to be taken when submitting a Spark job. As seen in the shell command above, there are two options, files and conf. The latter loads the file on the driver and the first transmits it to the executors. On each application execution, driver and executors write automatically their metrics in the carbon line protocol and they have the following form:

```
spark.application_1488980457743_0001.driver
  .jvm.pools.PS-Eden-Space.max
spark.application_1488980457743_0001.1
  .jvm.pools.PS-Eden-Space.max
```

Each line is a single measurement in the database, and considering the number of the metrics, it’s crucial to re-format these entries into a more manageable and familiar table form (section 4.2). The lines are dot separated with the first entry being the prefix, which will be used to identify the incoming Spark metrics from all the Graphite incoming traffic. The second item is the application id, that can take either the driver value or a numeric one to show the executor’s id and the last three, comprise the name of the metric.

Following the same logic, YARN and HDFS use the same file with their sources being more limited than the Spark ones, as they contain much more sub components which are not project relevant, such as jobtracker, tasktracker, maptask, reducetask etc (see Listing ). Compared to Spark, there is no setting need for setting up sources as each entity, for example nodemanager, will write its own information along with the jvm ones.

Finally, Telegraf is an agent program, part of InfluxData’s technology stack. It covers the role of data collecting, processing, aggregating and writing from a
machine throughout long periods of time. The developers have designed it to impact the machine’s memory as little as possible with several reports of CPU overuse[cite]. It’s biggest advantage is the support for a large number of plugins, home brewed and third party, covering the input and output but also processing and aggregating. The last two are only limited to just 1 plugin each but the former two are very rich in their capabilities. In this project the input plugin in use is the system(cpu, memory, network, disk, kernel etc) and the output, as expected, InfluxDB.

## 4.2 Database schema

In the InfluxDB section was mentioned that series are produced by all permutations of tags. This means that when a tag is highly variable, like a UUID, random string like a hash, salts etc lead to series with high cardinality that is one of the most common issues when the RAM usage is high[55][56]. For example, for machines with memory constraints, the developers need to consider the tag variability and even potentially turning tags into fields. On the other hand, if the data doesn’t have any tags at all, the query engine cannot reach its optimal speeds. Encoding metadata in tags is a good, and sometimes necessary, practice but it must be used carefully. In this project, a reason that played a vital role, was the usability of the queries. A query to retrieve data from a high number of measurements entails running regular expressions on them, as shown below:

```
spark.app_22222_1.driver.heap_used
spark.app_11111_1.1.heap_used
spark.app_33333_1.2.heap_used
```

```sql
SELECT "heap_used" FROM /.*app_22222_1.*/
```

In contrast, having a tabular structure means that the queries will resemble SQL, like so:

```
SELECT "heap_used" FROM "spark" where appid="app_22222_1"
```

The Graphite plugin on InfluxDB is highly customizable allowing for choosing a database, interval to flush data and control the size of data that are written each time, batches in memory etc. By default, the batch size is 1000 and having a timeout of 1 second means that data are flushed when one of the criteria are satisfied. Except of parsing the line protocol, the plugin allows template matching to filter incoming queries, extract tags from them and store them into
A special list called templates, initialized in InfluxDB’s properties, contains the filters and templates[57]. If the template is only one then a filter is not needed. When multiple filters are needed though, an incoming line runs through them with the stricter one winning and its template gets applied on the data before being saved. A template transforms the matching parts of a metric name into tags that are stored as tag keys and the value, if existing, is extracted from the corresponding part of the incoming line, otherwise the tag key is skipped.

Basic matching

input: servers.localhost.cpu.loadavg.10
template: .host.resource.measurement*
output: measurement=loagavg.10,
tags= host=localhost
resource=cpu

Multiple tags

input: servers.localhost.localdomain.cpu.cpu0.user
template: .host.host.measurement.cpu.measurement
output: measurement=cpu.user
tags= host=localhost.localadmin
      cpu=cpu0

Multiple measurements

input: servers.localhost.cpu.loadavg.10
servers.host123.elasticsearch.cache_hits 100
filter: servers.* - matches both
      servers.localhost.* - matches first

The plugin doesn’t come without its limitations though. When incoming data contain encoded data, for instance .host=dn0, then the templates cannot keep the value dn0, so a tag variable hostname would be set to the value between the dots. Another issue comes up when for the same measurement the number of dots varies, in which case there must be more specific filters for each case. Both of these problems arises due to the lack of regular expressions support. This affected metrics coming from YARN and HDFS as the number of dots would vary between 6 and 8 depending on the source. The solution to this was to follow the above advice and write multiple filters and templates for each special case:

...
4.3 Metrics retrieval

The Hops platform except of the Hadoop fork, it includes an abstraction on top of that, named Hopsworks, to simplify the user’s interaction with the system. Essentially, it breaks down to the backend and the frontend. The former is responsible for interacting with the databases, e.g. NDB and InfluxDB, to manage users, projects, ACL, file metadata, Spark client submissions via YARN, HDFS, ADAM etc. Following the popular microservices design, the backend exposes REST endpoints that cover a wide range of services for consumption by the web frontend. For instance, when a user navigates to the job’s Grafana user interface, one of the first endpoints, /appinfo, will be triggered to retrieve information about the application including the job’s start time, end time if the job has completed and number of executors. The base url that is going to be used onwards has the following form:

```
http://<servername>:<port>/hopsworks-api/api/project/<projectid>/jobs/<yarn_applicationid>
```

But not only Grafana uses it. In our developed system, it is a crucial endpoint as it provides the essential startup information, retrieved from InfluxDB, for our user interface. Except of the job’s start/end time and number of executors it also sends a list about each executor’s environment, i.e. host name, YARN container ID and the available VCores of the hosting YARN NM. It should be noted that the number of VCores is a YARN setting, so it is enough to query the database once. Each one of the previous values is essential in the functionality of our application.

The above endpoint is hosted by a service called JobService.java, already existing in the backend codebase. When the HTTP request reaches the service, it has already passed through authentication but not the authorization part, a bullet point in the future extensions chapter. Under this service, runs a new one that exposes the metrics by using the java client of InfluxDB, specifically /influxdb. Protecting an exposed database can be tricky, as there are many potential security loopholes and attack schemes, e.g. SQL injection as InfluxDB supports the
execution of subqueries. As a first, and only, step towards securing the endpoint, in the scope of this project, is for the java client to use a user that has only read rights, only for the relevant databases – graphite and telegraf.

InfluxDB’s query language resembles that of SQL and for the java client to send a query to the database a string parameter must be passed. One way to construct this is by just not constructing it and letting the frontend do the work, with the backend being just a worker. Therefore, most of the project’s complexity falls onto the frontend. A cleaner way would be to create an endpoint for each measurement, e.g. spark, nodemanager, but this limits the flexibility as there must be a GET optional value added for each special case. As the implemented solution, we chose to take out the variable parts of the query and share the complexity with the frontend. On an interval, the frontend updates the graphs, sending an HTTP GET request for each to basuUrl/influxdb. The REST API parameters that make up the query are:

```plaintext
select <columns> from <measurement>
where <tags>
group by <groupBy>
```

These variables are translated into a GET request that adheres to the following pattern:

```plaintext
/influxdb/<database>
?columns=<columns>
&measurement=<measurement>
&tags=<tags>
&groupBy=<groupBy>
```

The `columns` variable can be one or more names of an InfluxDB series or an aggregation function call, e.g. `mean(total_mem_used)`, `max(cpu_percent)`. A measurement is the equivalent of an SQL table and the tags among others include a time bound. It plays a very important role as it will limit the information travelling on the wire to these that arrived after the last call. Even though the retrieval will happen on an interval, a problem arises when deciding on how to determine the time ranges. One choice would be to statically increment a variable on each call, e.g. `X = UNIX epoch time` and for the next response `X = UNIX epoch time + interval`, but this has the potential risk of missing measurements as the network request is not instant.

On the other hand, if the kept time is the measurement of the last metric, then it is guaranteed that on the next backend call, the database will reply with all the newest, i.e. `X = time of last measurement of the latest response`, use this time as
the starting time for the next request. As far as the upper limit of the time range is concerned, in the case that the application is still running we will use the \textit{now()} timestamp, that will expand to the current UNIX epoch time on the database, or use the end time from the \textit{appinfo} end point, in case the application has terminated.

Example of parameters passed to influxdb endpoint

```plaintext
database = graphite
columns = max(threadpool_completeTasks)
measurement = spark
tags = appid=27application_1494328502680_0016 and
time > 1494925264000ms and
time < 1494925667313ms
groupBy = time(10s), service
```

4.4 Architecture

At this point, the metrics are being written and transformed automatically, to a tabular form, in InfluxDB. Also, in 4.3 we covered how the backend exposes a REST API available for getting consumed by the frontend services, although the details on how the backend extracts this information from InfluxDB are omitted as they are too low level\cite{58}. Nevertheless, this section brings together all the previous sections in this chapter, to compile a full picture of how the project is designed (Figure 4.2).

As mentioned before, this project will be an extension to the already existing codebase of Hopsworks and accessible through a web frontend. By providing the backend’s functionality through REST web services, it provides a platform agnostic data source that can be hooked into any interface the developers wish. In this case the frontend is built in AngularJS\cite{59}, or Angular 1, a modular and powerful javascript framework powered by Model-View-ViewModel\cite{60}, descendant of Model-View-Controller(MVC), architectural design pattern\cite{61}.

4.4.1 Angular JS

Building a complex website with dynamic elements, event delegation and in a concise way is not a simple undertaking if one is working on the vanilla web technologies like HTML and JS. With the booming popularity of information, web has been a vital component in spreading and sharing ideas and developing businesses. One of the biggest frameworks for building single page web applications is, as mentioned in the paragraph above, Angular JS developed by
Google, the open source community and corporations. It acts as an extension of HTML, since the latter is suitable for presenting static pages.

The first step of an Angular application is to parse the HTML template to detect additional custom tag attributes that get transformed into directives. They, in turn, handle the input and output parts of the page binding them dynamically with the model, that comprises a set of, automatically synchronised, javascript variables, that results into the client handling templates instead of the server. As an example, let’s assume that in the webpage there’s an input with an attribute called `ng-model="name"` aka `<input type="text" ng-model="name">`. Any change done through the page will be reflected automatically to the variable called `name` and vice versa.

In large applications, limiting the separations of concerns and business logic handling is essential in designing a scalable and maintainable application as more features are added. The variables mentioned in the previous paragraph, or the application’s state, i.e. the model, is stored in the controllers and their scope is declared as a directive in the page’s mark up. Following the recommendations of the AngularJS developers, a controller only contains business logic responsible for initializing and adding behavior in the child scope of the wrapped HTML view, as shown in Listing 4.3.

To wrap up this brief overview, the AngularJS services complement the controllers by taking over all the functionality that can be factored out in a common module and reused across the application. For instance, to call a REST endpoint of a server, the controller would call the build in HTTP service directly or
4.4. ARCHITECTURE

Figure 4.2: Architectural overview

through an application’s service that would act as the request delegate, potentially building up URL arguments passed from the function call. The built-in services cover a wide range of functionality, such as logging, running time intervals and accessing the root scope. They are injected in the controller code, following the modular and decoupled philosophy of the framework.

This project uses controllers to create separation lines between each, under monitoring, element of the distributed technology stack, that is the Spark driver and executors, Spark streaming information, metrics for worker machines and a general application overview. These controllers use a common service to share static information i.e. Spark application identifier and Hopsworks project identifier and expose functions for retrieving metrics from InfluxDB and the Spark REST endpoints.

4.4.2 Reading from the endpoints

Interacting with the database from the frontend is achieved through a service that is getting injected on each of the controllers. It’s functionality was developed in the premise of providing an easy and expressive way to read metrics. As mentioned above, the information shared is not complex and few in number as there is not much need of interaction between our projects’ controllers.

Other than the necessary getters and setters, there are functions that return a javascript promise. When a function call is of asynchronous nature, as in this case the HTTP GET requests, it is not an option to block until each one has received a result or failed. Angular provides the http service that returns a proxy object which the caller can subscribe to until a response has reached the frontend that will supply a result at some point in the future.
One of the functions, called `getMetrics`, handles requests related to InfluxDB. The arguments passed to it are exactly the ones that were described in Section 4.3. Recall that the group by argument is optional, even though in most of the graph queries it is supplied, and as such this conditional case is handled by this function. The other family of requests, includes these functions that call the Spark REST endpoints which cannot be factored into one since each endpoint needs different arguments[62].

In this project, we implemented a call to only one of the given endpoints but it provides essential information about the application’s state(see Table 4.1), the `/allexecutors` endpoint, which includes the driver along with all the running and executors in a final(finished, failed, killed etc) state. As shown below, the call to the backend is straightforward and the response comes in JSON format, an easy to work with data structure:

```javascript
getAllExecutorMetrics: function() {
    var query = '/api/project/' + self.projectId + '/jobs/' +
                self.appId + '/influxdb/allexecutors/' +
                '?filters=totalShuffleRead,totalShuffleWrite';
    return $http.get(query);
}
```

### 4.4.3 Controlling the views

A view in the web world can be described as a hierarchical tree construct that makes up what is displayed on the user’s screen. It plays a vital role in breaking down the business logic accordingly in a way that makes sense and does not hinder development and maintenance. To break down the views with perfect accuracy is a matter of experience and iterative reflection. Our developed user interface is composed of generic application information such as name, running time and application identifier and a tab for each monitored element, as described in the end of Section 4.4.1, being an independent entity and this leads to the creation of a controller for each one(Figure 4.3).
The metrics retrieval and the graph updating logic of each controller is very similar among them and as such the process can be described in an abstract manner. The tab’s content is only active, in the sense that it consumes resources such as memory, CPU cycles and sending requests to the backend, when it has been clicked on. This fact naturally allows for a controller to follow this lifecycle and update the graphs only while the graphs of the tab are visible.

As soon as a controller has been activated, as in Listing 4.3, the init function is called to set the graph data structures to their initial configurations, e.g. line color, empty values, graph name, and the controller’s internal state, and set up the running environment of the graphs, such as retrieving application(executors and host machines) information from the /appinfo endpoint. But this information, during an execution has a high chance of being invalidated, meaning that executors can dynamically be readjusted on any machine on the cluster, application might complete execution etc. To be aware of these changes, the initializing function sets up an interval timer that will do the same REST call as above. The timer frequency depends on the desired granularity, i.e. how fast does the developer want the UI to be aware of any cluster change, with the default value being 2 seconds.

At the same time, an updating timer is started that will refresh the graphs on a longer, compared to the /appinfo’s interval, by default 10 seconds. On timeout, a function will be triggered that takes care of calling each graph’s, or in general an interface element’s, updating function. Each visual element works independently of the presence of the others, as for each one we maintain separate state variables. For graphing the metrics, there were two choices, to request all of them on every update which would be very simple to implement as the frontend
would be essentially stateless but paying as a trade-off for increased network traffic. By keeping track of the time, as mentioned in Section 4.3, the queries and the business logic for updating the boundaries becomes more complex, on the other hand the information travelling on the wire is reduced to just new metrics.

The updating functions adhere to a template procedure. Other than the last measurement timer for each graph, there is also a boolean map, one entry per graph, that checks whether there has been at least one successful data load. A simple condition check at the top of the function, i.e. if the app is not running anymore and that data have been loaded at least once, the updating can be skipped as there will be no more relevant metrics written to the database.

At the core, the query filter, e.g. the *where* segment, along with the rest of the parameters are fed to the *VizopsService* that requests asynchronously the latest metrics. In case there are new metrics, the backend responds with *HTTP 200 OK*, which is a necessary condition to check before doing any updates on the internal
4.5 Graphing library

Most of the work will be part of developing the user interface and the underlying frontend architecture, making the choice of the graphing library an important choice. One may search on the internet and the results will not be surprising, a lot of approaches to charting in javascript, each with its bringing its own flair and philosophy, e.g. simple, complete, easy to extend. On top of that, with HopsWorks being developed in AngularJS, there’s a fork that has to be considered. Either create HTML directives that will abstract the graphing framework or use one that is already angular compatible. The former allows freedom of choice with the added directive development complexity and the latter inhibits the non-angular library by not having implemented all of its features. To avoid complexity and not reinvent the wheel, the second path is more practical.

There are two popular ways that libraries use to draw visualizations on the web, SVG and Canvas. Canvas was first introduced in the fifth, and current, version of the HTML standard[63]. By simply adding a canvas directive in a webpage, the programmer has a painting panel to manipulate how the pixels are drawn on it using javascript, usecases being rendering of game graphics, plotting, art and other visual images on the fly[63]. In contrast, SVG offers a way of rendering graphics through XML or scripting that manipulates a webpage’s DOM. It supports the displaying of vector graphics, text, images, graphical transformations(scaling, masking, rotating etc) and fonts among others.

Both approaches can achieve an equal level of visual fidelity, and as such the considerations boil down to the way each one draws the visual element and how it affects the user’s browser. SVG imagery manipulates DOM elements which has the advantage of being subject to events and user interactivity as any other element in a webpage. DOM based graphics have a drawback that as the objects to display increase, the rendering slows down and the memory fills up, but on a canvas approach, judging purely on rendering and not on the algorithms creating them, there’s no dramatic memory increase[64]. On the flip side, on larger rendering areas SVG keeps the rendering delays low whereas Canvas suffers a similar delay pattern as the SVG with large number of objects.
For our implementation, the graphs are expected to display a large number of points, which will induce larger rendering times but the benefits of out of the box interactivity, scalability to any resolution that will allow for the visualization to look crisp tipped the scale to an SVG based framework. The requirements that the chosen library should satisfy were numerous as shown below:

- **Lightweight graph rendering**: SVG is not as lightweight as the Canvas approach regarding memory;

- **Plethora of graphs**: The library should provide enough graphs to be expressive, even though graphs suitable for time series data are not that many;

- **Cheap partial redraws**: Adding new points in the graphs should not bring down the browser, even under many back end metrics requests;

- **Open source licence**: The licences of the libraries that to be used in the end should be released under an open sourced licence;

- **Developers & Community**: Having an active developer team and supporting community pushes for updates and fixes faster;

- **Technologies**: Should be built on the latest web technologies and standards;

- **Dependency chain**: Have short chain of dependencies that will keep the updating issues as little as possible;

- **Look & feel**: After all, the end user experience counts;

Taking into account the pros and cons of either approach, the end choice was to use a library based on **D3.js[47]**, with considerations for **Chartist.js[65]** and **plotly.js[66]**. It is a data driven javascript library that is based on open web standards, i.e. HTML, CSS, SVG and JS, that applies transformations on the webpage’s DOM to create visualizations with a focus on data. For our use it is fairly low-level and it would require a lot of effort from our side to implement a graphing system. **NVD3[67]** provides an abstraction layer on top by building charts and a builder pattern approach on setting various options for them, e.g. width, y axis label, x axis values transformations. The last building block of the framework abstractions is a library, **Angular-nvD3[68]**, providing **AngularJS** abstractions by initializing graphs through JSON objects.
4.6 Dashboard

The building blocks comprising our solution assemble into our application’s user interface, called Vizops. The view is organized into semantically grouped graphs and other visual information, such as how many executors and hosts are active at that time, total shuffle read/write bytes. Each controller is active only if the respective tab is active, except of the one handling the header of the interface, i.e. containing the time, name etc. Figure 4.5, depicts a streaming job that has been executed for almost 13 hours. In general, only two controllers out of the six will be running at any point, in this case GeneralCtrl and OverviewTabCtrl.

Except of side information, such as application name and identifier, the GeneralCtrl handles only a few, but important, visual elements. The clock is updated by a clock service. This clock monitors the appinfo timer interval, shown in Figure 4.4, for the application’s termination flag in which case the text displays the start and end times along with the execution duration. A selection box allows the user to choose an interval to group the measurements by[69], for instance when displaying metrics of an application running for a few hours, grouping the metrics every 30 seconds provides more detailed graphs but it costs in visual noise and a slowed down web browser. Optimally, the initial grouping interval should be chosen automatically for the user since in our first implementation, loading a multi-hour/day application will retrieve information with the default 10 second grouping interval.

The thinking behind the overview screen, the one that initially shows up when the interface launches, is to display information that will give a general impression about a live or completed application’s performance. The tasks are a common processing denominator of an application, as they will do the necessary computations so there can be progress. Knowing the completed tasks and the task completion rate, one can see any disruptions in the service that need further investigations and get a hint for stragglers as the rate will be almost horizontal. Additionally, the total active tasks, given that the zoom allows it, shows the shuffles between stages, AM restarting, stragglers.

As a first indicator of memory under utilization or starvation, there is a graph showing the total available memory of the containers and its used average heap memory. In conjunction to this graph, a user can get more analytic memory usage from the executors tab, which will be discussed below. HDFS read and write amounts are important as they allow to verify that the IO flow is as expected and that there are no issues arising from IO issues such as full disks and crashed machines as there would be anomalies in the graphs. The last piece is the top row, that displays the current number of executors, useful when an application uses dynamic executors, the virtual CPU utilization among others.

The driver, being the coordinator of the whole application, requires a tab on
its own. Depending on the workload’s requirements, initially unknown in some cases, memory tuning is one of the crucial resources to monitor. Low maximum memory leads to a lot of time spent in GC, as explained in Chapter 3 or even Out of Memory exceptions. To avoid recomputing RDDs, a programmer has the choice of caching them into a dedicated section in the memory which is handled by the BlockManager service. In cases that the cache is not enough, the RDDs will still be persisted but on the disk this time. Network can potentially be a bottleneck since the driver has to send out information to executors. A user, through the Hosts tab, can filter the network traffic of the driver’s machine, a name that can be obtained from the driver’s tab.

The complement of the driver, the executors, take up the responsibility of executing the tasks assigned to them. Ranging from only one to thousands, it can get out of hand quickly when attempting to monitor their status and their environment, e.g. YARN containers. Spark’s web interface provides aggregated and per executor information but not in an easily digestible manner. In our implementation, the default view is to display metrics aggregated across all the executors but there’s the ability to view the ones running one a single machine, a potential culprit, or to view only a single one by using its identifier, an integer number.

As described in Chapter 3, data skew is a reoccurring problem. The interface displays the task distribution per executor in a bar chart and in conjunction with the shuffle read/write per executor, also a bar chart, a user can easily identify
skewed input data. Monitoring the memory status as well as the GC is important, for this reason we have included a bar chart to display the peak heap memory per container JVM to determine, with the assistance of the average memory graph, if they have enough. Executors working on maximum memory means that the not fitting data will be spilled to the disk/HDFS, which can be viewed in the respective read and write graphs additionally to hints that the application may be IO bound. Last but not least, when the application is computationally intensive, utilizing the available virtual CPU is essential to get the most out of the resources.

The hosting machines may be the root of slowdowns affecting all the hosted containers. This, filterable by hostname, panel aims to give insight to the user on the load of the machine through a physical CPU graph that displays the used, IO and idle percentages, as collected from the Telegraf daemon, with the second graph being about the memory usage. In some cases, a user might notice that the application’s write operations max out and that might originate from the machine’s disk being full. As we discussed above, an application, especially a streaming one, that makes heavy use of the network, might slow down due to an under load machine. To further provide information about the task and executor distribution and consequences of that we added two graphs showing the executors and tasks per physical machine.

Modern clusters, run hundreds or thousands of machines, spreading an application’s execution across. Both the executors and hosts page, since they support filtering per host, will be slowed down due to the fact that there’s no logic in this version of the implementation to display them in a better designed user interface element.

Finally, a streaming Spark application introduces new metrics to monitor and exploit to gain insight. When a mini batch is filled up, it is queued up until picked up for processing. This delay, scheduling and processing, will cause backpressure and lag to the pipeline in the event of it being higher than the batching interval. The effect of such pressure building up, is the difference between received and processed batches at any time. Having a big gap without a convergence tendency, translates to input lag which might be unacceptable in certain use cases.
Chapter 5

Evaluation

Throughout this work, it was discussed how the controller interactions have been optimized to minimize the network traffic and the requests overall. In this section the developed application will be assessed on both quantitative and qualitative basis through a few simple scenarios in a small cluster setup. The cluster consisted of three virtual machines, 2 physical cores with 2.6 GHz frequency and 15GB of memory each, deployed using the automated procedure of the Hops platform.

The conducted experiments were based on executing a CPU computationally intensive application, Pi approximation using 30,000 iterations. The following scenarios will be put under test:

- Restart the NM running on one of the three nodes with a brief sleep between the stop and start;
- Tune the job for maximum, limited by the machine’s resources, cluster CPU utilization and time combination;
- Analyze the executor’s memory usage to assess whether the allocated container memory is optimal for the application’s nature;

A YARN NM crashing during execution can have dire effects on the application’s execution, taken that there were relevant executor containers on that machine. Through Spark UI, the user would see that executors terminated and the task completion dropping, depending on the loss. An experienced user potentially could isolate the problem by going through the logs. Figure 5.1a identifies the issue immediately, with two major drops in the graph, one signifying the NM crash and the latter, the container re-allocation. Filtering the graphs, figures 5.1b and 5.1c, by host it is clear that it was hopsworks1, rest being hopsworks0 and hopsworks2, whose NM crashed. This quick detection of the issue saved investigation time and the tracing of events can be focused on investigating the
faulty node.

For the next scenario the application under execution was the same as in the previous one. The containers allocated the same memory, 768MB, and the number of virtual cores was fixed to 2. Figure 5.1d shows the peak memory of each executor throughout the execution and clearly shows that the reserved container memory is not required since this application does not read input data or trigger expensive shuffles.

A recurring process is that of tuning a job to exploit to the maximum the resources available. This scenario focuses on tuning the toy application, exploring a variable number of virtual cores, capping at 8 per container, in conjunction with Spark’s dynamic execution feature. The parameter when disabling dynamic execution becomes the number of executors itself but in this case the virtual cores do not change throughout the evaluation.

Both tables 5.1 and 5.2 contain information as extracted from Vizops’s interface. Having prior knowledge that the application is computationally intensive but light on memory usage since there’s no IO or intermediate data, the results verify that the more virtual cores available to each container the better the performance, meaning that Spark Pi is CPU bound. By keeping the virtual
Table 5.1: Spark Pi 30,000 iterations, Dynamic execution on

<table>
<thead>
<tr>
<th>Dynamic execution</th>
<th>Time</th>
<th>VCPU util.</th>
<th>Allocated Executors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 core</td>
<td>10:20</td>
<td>6.3%</td>
<td>14</td>
</tr>
<tr>
<td>2 cores</td>
<td>08:06</td>
<td>9.5%</td>
<td>11</td>
</tr>
<tr>
<td>4 cores</td>
<td>09:19</td>
<td>12.8%</td>
<td>5</td>
</tr>
<tr>
<td>8 cores</td>
<td>05:15</td>
<td>55.1%</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.2: Spark Pi 30,000 iterations, Fixed # executors, 2 cores

<table>
<thead>
<tr>
<th>Fixed executors</th>
<th>Time</th>
<th>VCPU util.</th>
<th>Allocated Executors</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 executors</td>
<td>07:44</td>
<td>31.3%</td>
<td>2</td>
</tr>
<tr>
<td>4 executors</td>
<td>07:06</td>
<td>21.8%</td>
<td>4</td>
</tr>
<tr>
<td>8 executors</td>
<td>09:50</td>
<td>8.4%</td>
<td>8</td>
</tr>
<tr>
<td>16 executors</td>
<td>07:47</td>
<td>9.5%</td>
<td>11</td>
</tr>
</tbody>
</table>

cores fixed and modifying the executors count, a declining trend appears in both time and VCPU utilization. Concluding, we see that by increasing the number of cores and/or reducing the number of executors, in the general case, improves the application's execution time and core utilization. This happens due to the cost of operations adding overhead, such as scheduling and distribution of tasks which agrees with the common phenomenon of applications not benefiting from an increased level of parallelism.
Chapter 6

Conclusions & future work

Wrapping up, in this chapter we discuss the contribution of this work and reflections upon it, future potential ways of extending the application and limitations of this work.

6.1 Conclusions

The visualized metrics coming from a variety of sources, participating in a multi-tenant distributed environment, enabled users of Hops to detect issues and optimize their data processing pipeline. Having a tool such as Vizops, the Spark developer gains holistic knowledge through views ranging across the application itself, the hosting containers and the physical cluster machines. Its dual component, the Spark web interface, is an excellent source of fine-grained details about each application’s component, e.g. jobs, stages, tasks, executors and more, that aids in the developer’s investigative analysis.

While developing Vizops one of the architectural design requirements was to build a modular codebase that would constrain the metrics processing complexity for each graph to the updating procedure. On top of that, the user interface follows this thinking with the tabular view which is easily extensible with new services to monitor. As long as a framework can write metrics to InfluxDB/Graphite or expose REST endpoints, the choices are quite vast.

6.2 Limitations

In the given master thesis timeframe, we developed a reliable visualization tool that given more effort and resources could be extended with immensely more functionality as presented in 6.3. Storing metrics on a database costs storage
space, and taking into account number of potential users, it can be a bottleneck. The data on InfluxDB in the current cluster configuration are stored for a week and subsequently removed. This sets a hard cap on the nature of jobs and their depth, as the user will be able to see only a few days back.

On a more abstract basis, the choice for focusing on Apache Spark, is one of the frameworks, that are supported by Hops and the most popular among them. Being a popular tool meant that there would be more discussions and reflections on production setups, fact that served as a precious source of information to identify key metrics and to build correlations between them.

6.3 Future work

As Spark matures the need for sane monitoring tools becomes an issue of utmost importance, with our implementation laying the groundwork for future extensions. The latest addition of GPU support as a YARN resource on Hopsworks[70] allows for Tensorflow applications to run on Spark or directly on YARN which can be monitored through our visualization panel as a new tab or integrated into the already existing ones. Other than Spark, Flink supports metric dumping to graphite, making it a strong candidate for new visualizations.

Except of GPU monitoring, there’s a myriad more information that can be visualized such as more information from Spark REST endpoints, monitoring Kafka queues and consumer lag, status notifications using Kapacitor[71] for important events coming from HDFS, NDB, telegraf plugins or even from more complex component interactions.

The user experience with the application can also be improved by making the user interface more flexible by allowing the re-positioning and showing/hiding of graphs. On a more important note, graphs need to display the information in a clean and easy to digest manner which can be achieved by removing noise and smoothing out the data[72].
Bibliography


[19] *Apache Hadoop 2.0 and Yarn - Figure*. July 31, 2017. URL: https://www.edureka.co/blog/apache-hadoop-2-0-and-yarn/.

[21] *Apache Hadoop YARN - ResourceManager Components - Figure*. July 31, 2017. URL: https://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/.


[23] *Map Reduce Execution - Figure*. July 31, 2017. URL: http://blog.cubrid.org:8080/?mid=textyle&category=dev-platform&alias_title=platforms-for-big-data&vid=blog&l=ko.


[70] YARN GPU Support on Hops. July 11, 2017. URL: https://github.com/hopshadoop/hops/tree/elef6c178695e4fff0c3515c5c9e109bf8e5ad.


[72] Automating Dashboard Displays with ASAP. Sept. 3, 2017. URL: https://speakerdeck.com/player/e2c4c4a1133c4cc899504faf35d8047b#.